

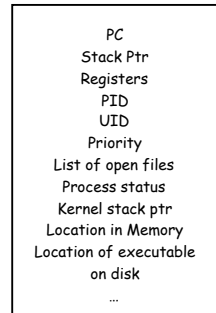
# Virtualizing the CPU

- OS keeps a PCB for each process
- It has space to hold a "frozen" version of the process's state

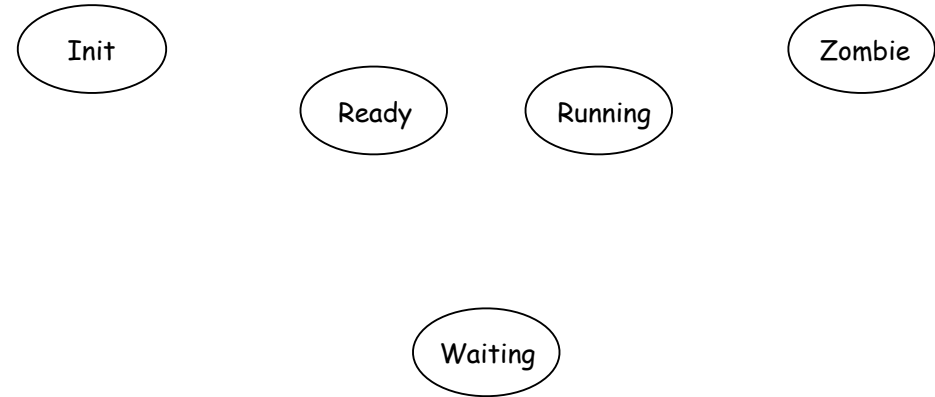
- Program counter
- Process status (ready, running, etc)
- CPU registers
- CPU scheduling info
- Memory management info
- Account info
- I/O status info

to be saved when the process relinquishes the CPU  
and reloaded when the process reacquires the CPU

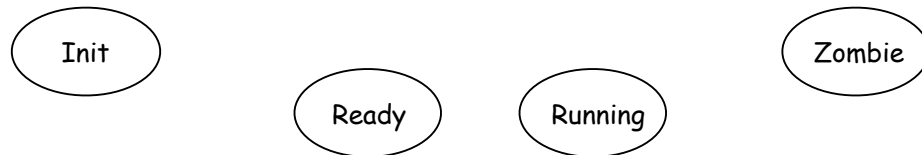
Process Control Block



# Process Life Cycle



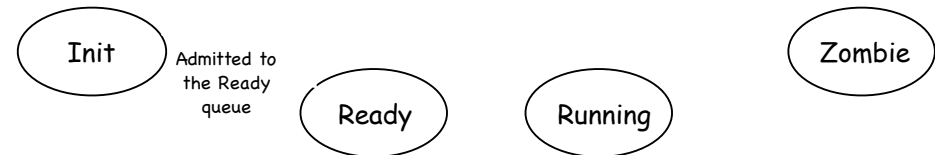
# Process Life Cycle



PCB: being created  
Registers: uninitialized



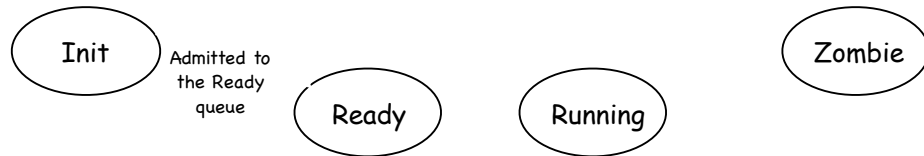
# Process Life Cycle



PCB: being created  
Registers: uninitialized



# Process Life Cycle

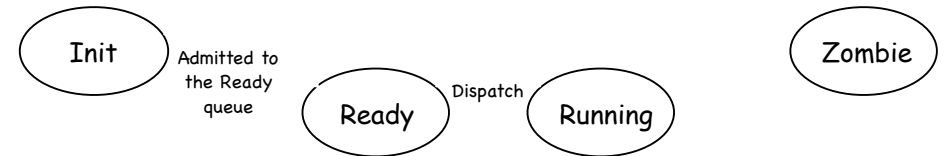


PCB: on the Ready queue  
Registers: pushed by kernel code onto interrupt stack

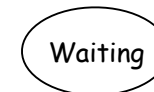


75

# Process Life Cycle

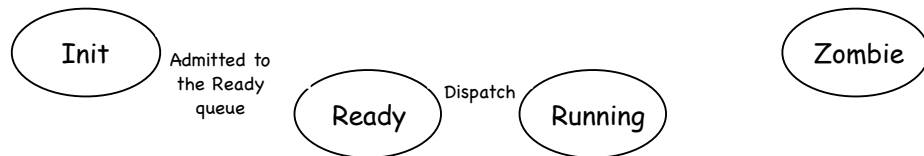


PCB: currently executing  
Registers: popped from interrupt stack into CPU



76

# Process Life Cycle

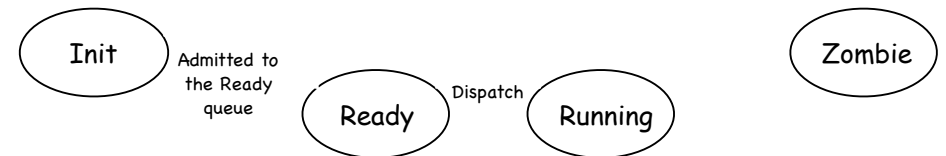


PCB: on Ready queue  
Registers: pushed onto interrupt stack (SP saved in PCB)



77

# Process Life Cycle

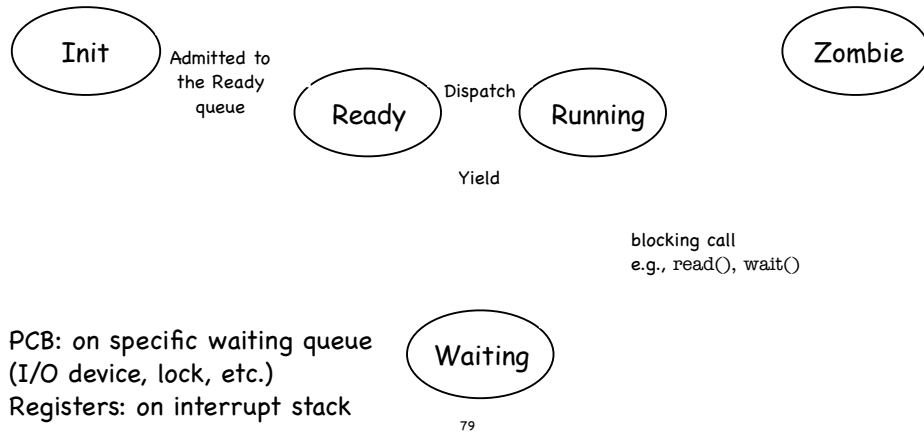


PCB: currently executing  
Registers: popped from interrupt stack into CPU

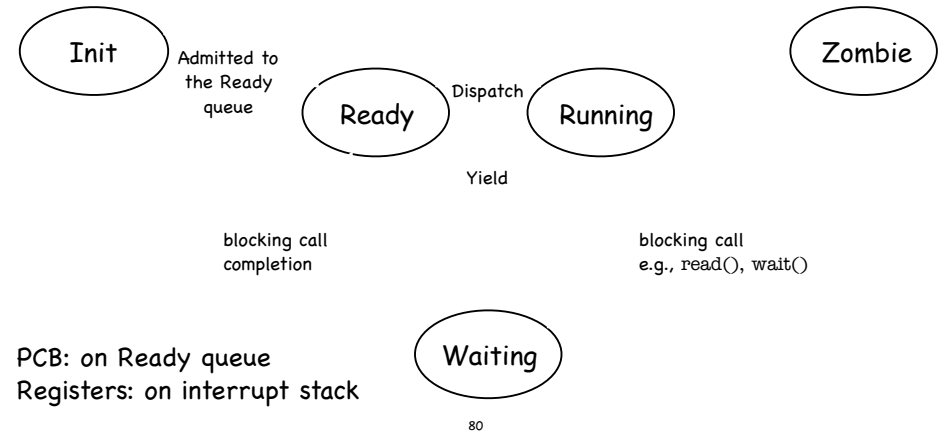


78

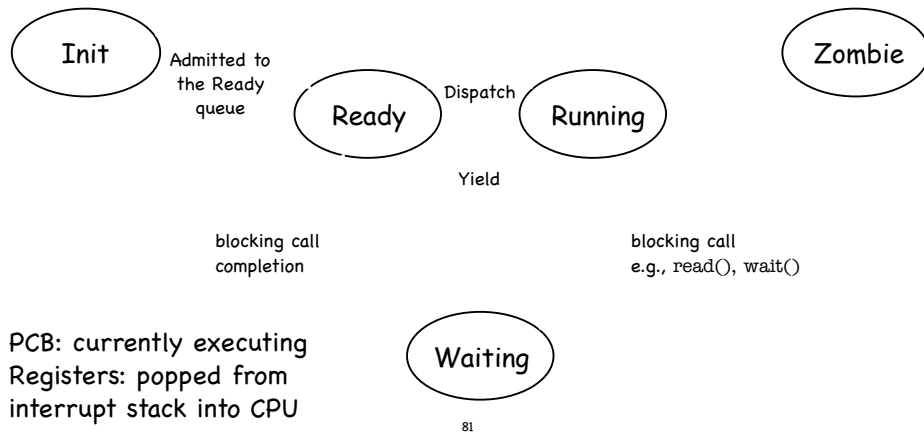
# Process Life Cycle



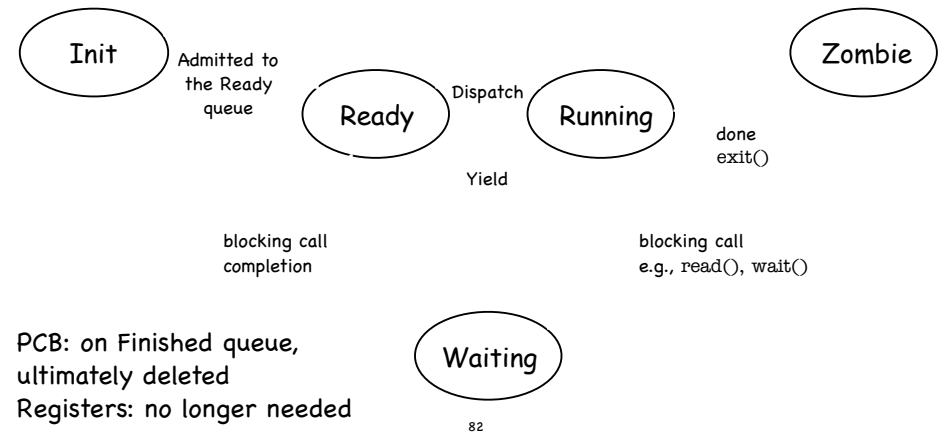
# Process Life Cycle



# Process Life Cycle



# Process Life Cycle



# Invariants to keep in mind

- At most one process/core running at any time
- When CPU in user mode, current process is **RUNNING** and its interrupt stack is empty
- If process is **RUNNING**
  - its PCB not on any queue
  - it is not necessarily in **USER** mode
- If process is **RUNNABLE** or **WAITING**
  - its registers are saved at the top of its interrupt stack
  - its PCB is either
    - on the **READY** queue (if **RUNNABLE**)
    - on some **WAIT** queue (if **WAITING**)
- If process is a **ZOMBIE**
  - its PCB is on **FINISHED** queue

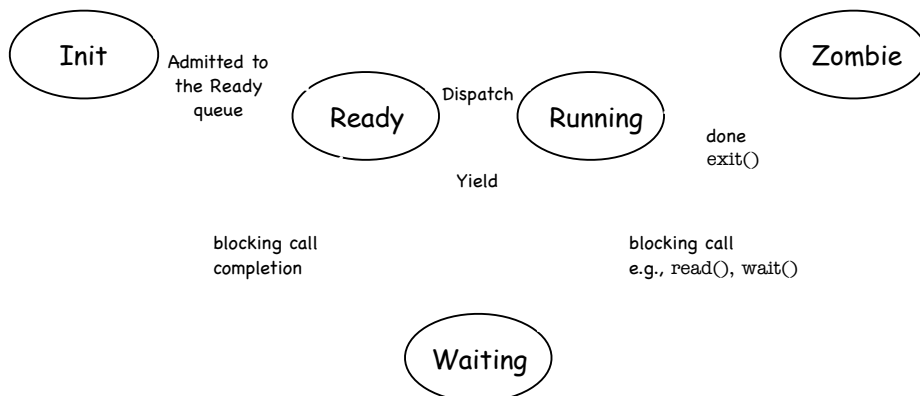
# Cleaning up Zombies



- Process cannot clean up itself (why?)
- Process can be cleaned up
  - by some other process, checking for zombies before returning to **RUNNING** state
  - or by parent which waits for it
    - but what if parent turns into a zombie first?
    - or by a dedicated "reaper" process
- Linux uses a combination
  - if alive, parent cleans up child that it is waiting for
  - if parent is dead, child process is inherited by the initial process, which is continually waiting



# Process Life Cycle



# How to Yield/Wait?

- Must switch from executing the current process to executing some other **READY** process
  - Current process: **RUNNING**    **READY**
  - Next process: **READY**    **RUNNING**
- Save kernel registers of Current on its interrupt stack
- Save kernel SP of Current in its PCB
- Restore kernel SP of Next from its PCB
- Restore kernel registers of Next from its interrupt stack

# Yielding

```
ctx_switch: //ip already pushed
pushq %rbp
pushq %rbx
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %r11
pushq %r10
pushq %r9
pushq %r8
movq %rsp, (%rdi)
movq %rsi, %rsp
popq %rbp
popq %rbx
popq %r15
popq %r14
popq %r13
popq %r12
popq %r11
popq %r10
popq %r9
popq %r8
retq
```

```
struct pcb *current, *next;

void yield(){
    assert(current->state == RUNNING);
    current->state = RUNNABLE;
    readyQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp)
    current = next;
}
```

# Starting a New Process

```
ctx_start:
pushq %rbp
pushq %rbx
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %r11
pushq %r10
pushq %r9
pushq %r8
movq %rsp, (%rdi)
movq %rsi, %rsp
retq
```

```
void createProcess( func ){
    void *SP;
    current->state = READY;
    readyQueue.add(current);
    struct pcb *next = malloc(...);
    next->func = func;
    next->state = RUNNING;
    SP = next->top_of_stack;
    * -- SP = PSW;
    * -- SP = USP;
    * -- SP = UPC;
    ctx_start(&current->sp, SP)
}
```

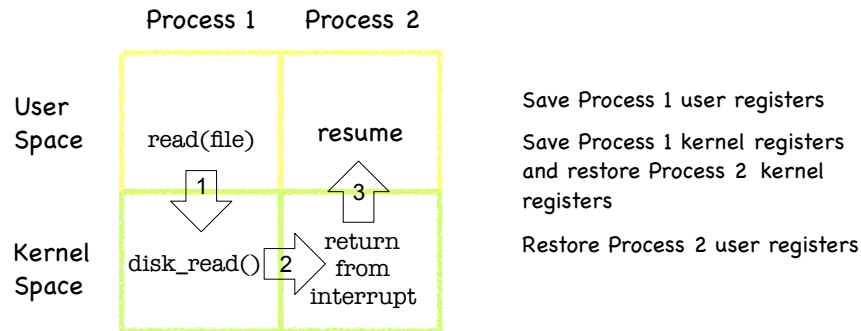
## Anybody there?

- ⦿ What if no process is READY?  
scheduler() would return NULL — aargh!
- ⦿ No panic on the Titanic:  
OS always runs a low priority process, in an infinite loop executing the HLT instruction  
halts CPU until next interrupt  
Interrupt handler executes yield() if some other process is put on the Ready queue

## Three Flavors of Context Switching

- ⦿ Interrupt: from user to kernel space  
on system call, exception, or interrupt  
Px user stack   Px interrupt stack
- ⦿ Yield: between two processes, inside kernel  
from one PCB/interrupt stack to another  
Px interrupt stack   Py interrupt stack
- ⦿ Return from interrupt: from kernel to user space  
with the homonymous instruction  
Px interrupt stack   Px user stack

# Switching between Processes



Save Process 1 user registers  
 Save Process 1 kernel registers and restore Process 2 kernel registers  
 Restore Process 2 user registers

# System Calls to Create a New Process

- Windows: `CreateProcess(...);`
- Unix (Linux): `fork() + exec(...)`

## CreateProcess (Simplified)

```
if (!CreateProcess(
    NULL,          // No module name (use command line)
    argv[1],      // Command line
    NULL,         // Process handle not inheritable
    NULL,         // Thread handle not inheritable
    FALSE,       // Set handle inheritance to FALSE
    0,           // No creation flags
    NULL,        // Use parent's environment block
    NULL,        // Use parent's starting directory
    &si,         // Pointer to STARTUPINFO structure
    &pi )       // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

## fork (actual form)

```
process identifier
int pid = fork();
```

..but needs `exec(...)`

[Unix]

# Kernel Actions to Create a Process

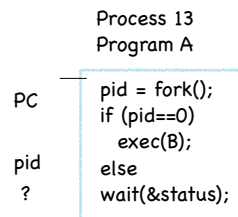
- fork()
  - allocate ProcessID
  - initialize PCB
  - create and initialize new address space
  - inform scheduler new process is READY
- exec(program, arguments)
  - load program into address space
  - copy arguments into address space's memory
  - initialize h/w context to start execution at "start"
- CreateProcess(...) does both

# Creating and managing processes

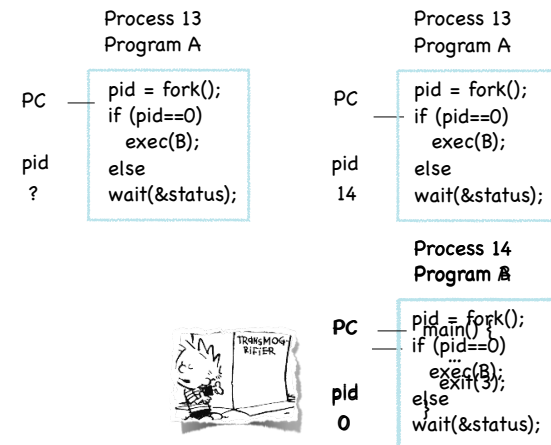
Syscall	Description
fork()	Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child
exec (prog, args)	Run application prog in the current process with the specified args (replacing any code and data that was present in process)
wait (&status)	Pause until a child process has exited
exit (status)	Tell kernel current process is complete and its data structures (stack, heap, code) should be garbage collected. May keep PCB.
kill (pid, type)	Send an interrupt of a specified type to a process (a bit of an overdramatic misnomer...)

[Unix]

## In action



## In action



# In action

